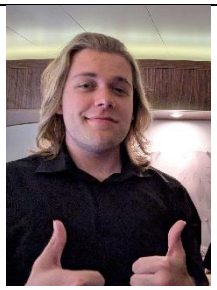
SP-104 Active Learning System Images

Software Test Plan and Report Document

CS 4850 – Section 03 – Fall 2025

Matthew Hall, Josh Smith, Elijah Merrill, Noah Lane

			
Josh Smith Team Leader Developer	Elijah Merrill Developer	Noah Lane Documentation	Matthew Hall Documentation

Team Members:

Name	Role	Cell / Alt Email
Josh Smith (Team Lead)	Developer	706-414-2827 joshuasmith0515@gmail.com
Elijah Merrill	Developer	478-283-0811 elijahmerrill04@gmail.com
Matthew Hall	Documentation	678-873-8542 thematthewhall7@gmail.com
Noah Lane	Documentation	706-591-2312 noahlane142@gmail.com
Sharon Perry	Advisor	770-329-3895 sperry46@kennesaw.edu

1.0 Test Objectives

The primary objective of testing was to verify that all components of the command line-based active learning system behaved correctly and worked together as intended. This included confirming that the dataset was loaded properly from the local file structure, that preprocessing steps were executed without errors, and that the model could train successfully using the provided configuration. Testing also aimed to validate that the active learning loop correctly selected new samples, updated internal data structures, and saved results to the appropriate output directories. Additional objectives included ensuring that all command line responses, logs, and generated files, such as CSV or Parquet index tables, accurately reflected the system's operations. Overall, the testing process focused on confirming that the system performed reliably in a terminal environment and produced consistent and verifiable results throughout each stage of execution.

2.0 Scope of Testing

The scope of testing for this project covers all functional components of the command–line–based active learning system, from initial data ingestion through model training, evaluation, and result generation. Testing focuses primarily on verifying that each module behaved correctly when executed in a standard local workstation environment and that the overall workflow produced stable, reproducible outputs.

The testing effort included:

- **Dataset Management:** Verifying dataset indexing, file path resolution, metadata extraction, and split generation by patient ID to ensure data integrity before training.
- **Data Loading and Preprocessing:** Confirming that images, labels, and metadata could be loaded efficiently on CPU and GPU; checking that transformations were applied consistently; and ensuring that batches produced expected tensor dimensions.

- **Model Training Workflow:** Testing the training loop for stability, correct loss computation, ability to run full epochs, proper checkpoint saving, and correct handling of command line parameters such as subset size and number of workers.
- **Evaluation Pipeline:** Validating that trained models could be evaluated without errors, that metrics such as AUC-ROC and AUC-PR were computed and logged, and that outputs were written to the designated directories.
- **System Performance and Hardware Utilization:** Confirming that GPU acceleration was detected and used where appropriate, assessing runtime behavior, and monitoring resource usage through external tools such as *nvidia-smi*.
- **Command Line Interaction and Logging**: Ensuring that all terminal outputs, progress bars, warnings, and log files accurately reflect system operations.

The testing did not include large-scale stress testing, cross-platform compatibility validation (e.g., Linux or macOS environments), model hyperparameter optimization, or verification against an external database or distributed system. Additionally, the scope did not extend to user interface testing beyond command line interactions, nor did it include testing of any future or optional features outside the core active learning pipeline.

Overall, the scope was designed to ensure that each component of the system functioned reliably on a single-machine setup and that the full active learning workflow could be executed end-to-end with consistent results.

3.0 Test Cases

The following test cases were developed to validate the functional behavior of the command line–based active learning system. Each test case targets a specific component of the pipeline to ensure correctness, reliability, and consistency across the data processing, training, and evaluation stages.

Test Case 1 – Dataset Indexing Validation
Ensures that the indexing module correctly locates image files, extracts metadata, and generates a complete, error-free parquet index. Verifies behavior when file paths are missing or malformed.

Test Case 2 – Patient-Level Dataset Splitting
Confirms that the dataset is correctly partitioned into training, validation, and test sets without patient overlap and that the resulting split sizes align with expected ratios.

Test Case 3 – Data Loader and Preprocessing Functionality
Validates that batches are loaded to CPU/GPU; transformations are applied correctly, and that each batch contains properly shaped tensors and valid labels.

Test Case 4 – Training Loop Execution
Tests that a full training epoch can run end-to-end without runtime errors, that loss values behave as expected, and that checkpoints are saved successfully.

Test Case 5 – Model Evaluation Pipeline
Checks that evaluation runs automatically after training, producing valid metrics (AUC-ROC, AUC-PR) and correctly generating logs and confusion statistics.

Test Case 6 – GPU Device Utilization
Ensures that the system detects a CUDA-enabled GPU, uses it during training, and exhibits increased memory and computer utilization during runtime.

Test Case 7 – Logging and Output File Generation
Verifies that all log files, CSV or Parquet outputs, metrics summaries, and checkpoints are generated in the correct directories with accurate and consistent content.

Test Case 8 – Error Handling and Graceful Failure
Checks system behavior when encountering missing pretrained weights, missing file paths, or invalid configuration parameters, ensuring that failures produce clear error messages rather than crashes.

4.0 Test Procedures

Procedure 1 – Verify Dataset Indexing
Goal: Ensure the indexer correctly maps image filenames to full file paths and loads metadata.

Steps:

1. Run the indexing script: python -m src.data.indexer
2. Inspect the generated parquet file (confirming columns: image_id, patient_id, view, label_raw, path).
3. Check for missing file paths.

Expected Results:

- Complete indexing with no errors.
- All image IDs have valid file paths.
- Missing or invalid paths are properly flagged as missing=True.

Procedure 2 – Validate Dataset Split by Patient
Goal: Confirm that patients do not appear in multiple splits.

Steps:

1. Run the splitter script: python -m src.data.splitter
2. Open the generated splits.json file.
3. Check that each patient's ID appears only in one of the sets: train, validation, or test.

Expected Results:

- Splits are generated successfully.
- No patient overlaps across sets.
- Counts roughly match desired ratios (70/15/15).

Procedure 3 – Verify Data Loader Functionality
Goal: Ensure images and labels load properly onto GPU/CPU and transformations apply correctly.

Steps:

1. Run the smoke test: python -m src.data.smoke
2. Observe printed tensor shapes via output.
3. Confirm each batch load without raising expectations.

Expected Results:

- Batch shape matches expected (N, C, H, W) dimensions.
- Labels match expected class count.
- No failures in loading images or applying transformations.

Procedure 4 – Validate Training Loop
Goal: Test that training executes one full epoch without runtime errors.

Steps:

1. Run trainer file for 1 epoch: python -m train.trainer –subset 100 –epoch 1
2. Monitor progress bar and GPU Utilization as well as the amount of time to process.
3. Verify that a model checkpoint is created.

Expected Results:

- Training loop runs without crashing
- Loss decreases at least slightly within the epoch
- The best model run is saved to the user's system.

Procedure 5 – Evaluate Model on Validation Set
Goal: Confirm the evaluation pipeline runs end-to-end and produces valid metrics

Steps:

1. After training, run the evaluation automatically produced by the trainer.
2. Check AUCROC and AUCPR values.
3. Inspect printed confusion values for any abnormalities.

Expected Results:

- Evaluation completed successfully.
- AUC values are within a plausible range (aiming to be higher than 0.5 for a functional model)
- Metrics saved and logged.

Procedure 6 – GPU Utilization Verification
Goal: Confirm that the system correctly detects and uses the GPU for acceleration

Steps:

1. Run the trainer, making sure it's using GPU (CUDA) for training.
2. In a separate terminal (Command Prompt, PowerShell, etc.), run nvidia-smi.
3. Watch for active GPU memory usage.

Expected Results:

- GPU is recognized as the active device and not the CPU.
- GPU memory and compute utilization increase during training.
- Training runtime is significantly faster compared to CPU-only runtime.

5.0 Test Environment

The test environment consisted of a standard workstation running Windows 11 with an Intel-based processor and 32GB of RAM, providing sufficient performance for running the active learning system and processing the dataset. All development and testing were performed using Python, along with required libraries such as PyTorch, Pandas, and NumPy. The system was tested in a local environment, without using a virtual environment to manage dependencies and ensure consistency across test runs. No external database was required, as the project relied entirely on a local dataset stored in structured files. Network usage was minimal, limited to local communication between components such as the backend API and interface. This setup ensured a controlled and reproducible testing environment throughout the development process.

6.0 Test Data

The testing process required a combination of input data, configuration files, software dependencies, and project-specific source code in order to execute all functional components.

Testing relies on a locally stored medical imaging dataset. Specifically, the Data_Entry_2017.csv file from Kaggle's NIH Chest X-rays dataset. This, along with the associated image files organized in the expected directory, provided the labels, patient identifiers, and paths needed for dataset indexing and model training.

All code was executed using the project's command line-based modules.

- Data Processing Scripts
  - indexer.py
  - dataset.py
  - splitter.py
  - smoke.py
- Training Scripts
  - metrics.py
  - trainer.py
  - model_factor.py
- Configuration Files
  - Config.yaml

The system also required an up-to-date version of Python and several core libraries to run correctly. These include PyTorch, NumPy, Pandas, TorchVision, and TQDM, which support tensor operations, image transformations, data loading, and progress monitoring. All testing was performed using Python 3.x on a Windows 11 workstation, ensuring a consistent and reproducible environment.

7.0 Software Test Report

Requirement	Pass	Fail	Severity
Dataset indexing completes without missing paths	X		Moderate
Train/Val/Test split produces correct patient counts	X		Minor
GPU device detection (CUDA available)	X		Critical
DataLoader loads batches without crashing	X		Critical
Model forward pass executes without errors	X		Critical
Training loop runs full epoch without interruption	X		Critical
Evaluation loop produces metrics (AUC, AUCPR)	X		Major
Progress bar displays correctly during training	X		Minor
Checkpoint saving creates .pt file	X		Major
Handling of subset parameters (ex, 2000 samples)	X		Minor
Support for multiple workers (DataLoader)	X		Major
Graceful failure if pretrained weights unavailable		X	Minor
Model achieves minimum acceptable AUC (>0.55)		X	Critical
Evaluation time under performance threshold		X	Moderate